# 板子

## 数据结构

### *链式前向星*

```
//链式前向星 是以存边的方式去存图的这么一种数据结构
//先定义一个边的结构体数组
const int maxn=100;
int cnt; //可以说是记录边的数量
struct Edge{
    int to;//指向的点
    int w;//边权
    int next;//指向的下一条边
}edge[maxn<<1]; //无向图要*2
int head[maxn];//用来记录节点的边的 链表 初值全为-1（可为其他） 后面会用到 链表的头插法
// for (int i=0;i<maxn;i++){
//     head[i]=-1;
// }
void add_edge(int u,int v,int w){//加一条 u到v的单向边
    edge[cnt].to=v;
    edge[cnt].w=w;
    edge[cnt].next=head[u];//因为是头插法
    head[u]=cnt++;
}
// 遍历 u的出边
for (int i=head[u];~i;i=edge[i].next[i]){ // ~i 即 i!=-1
    int v=edge[i].to[i];
}
```

### *邻链表*

```
//是用vector实现的 时间复杂度和上面一致
const int N=1e5;
vector <int> Edge[N];//vector <pair<int,int>> Edge[N] 加权写法 前一个int代表点 后一个代表权
void add_edge(int u,int v){//加一条 u->v的边
    Edge[u].push_back(v);
    Edge[v].push_back(v);//加双向边
    //带边权的写法
    Edge[u].push_back({v,w});
    Edge[v].push_back({u,w});//加双向边
    //无向图则再来一遍
}
//遍历一个点周围的点
for (auto i:Edge[u]){
```

```
}
```

# 并查集

主要是用到并查集的两个操作 合并 和 查询
最初先把每个点都看成是一个集合 然后他们自己的祖先就是它自己
然后合并操作 就是把这个集合的祖先 连到 另外一个集合的祖先就好了
查询 就是看这两个点的 祖先是不是同一个 如果是 则说明在同一个集合里 如果不是 则不在
代码实现

```
#include <bits/stdc++.h>
using namespace std;
const int N=2e5+4;
int f[N];
int Get_root(int x){
    if(f[x]==x) return x;
    return f[x]=Get_root(f[x]);
}
/*
void hebing(){//合并两个并查集的操作
    int x=Get_root(x);
    int y=Get_root(y);
    if(x==y) return ;  //这个很重要 一定不要忘记
    f[x]=y;
}
*/
int n,m;//n个点 m次操作
int op,x,y;
int main(){
    cin>>n>>m;
    for (int i=1;i<=n;i++) f[i]=i;
    while(m--){
        cin>>op>>x>>y;
        if(op==1){//合并两个集合的操作 也可以看成是连接
            x=Get_root(x);
            y=Get_root(y);
            f[x]=y;
        }else {
            if(Get_root(x)==Get_root(y)){//查询操作
                cout<<"Y";
            }else {
                cout<<'N';
            }
        }
    }
    return 0;
}
```

# 树状数组

```
//优点 可以一边修改 一边查询 都是log n的复杂度
//可以做到: O(logN) 询问区间和、O(logN) 单点修改。
//关于树状数组的初始化 是根据它的范围 用前缀和数组初始化
//先要介绍一个前置的东西 lowbit 用途是取一个二进制数字的最后一位1
int lowbit(int x){
    return (x&-x);
}
//那么查询就可以借助上面这个东西来解决
//先定义一个数组C让其中的每一个数字都掌管一个区间和 -> c[x] 掌管lowbit(x)的范围
//c[x]= [x-lowbit(x)+1,x]的所以元素的和
```

这样，要询问前缀和[1,x]的话，我们可以先用lowbit(x)得到前缀里第一个区间的和，然后用x减去lowbit(x)，重复操作得到其他的区间和。因为我们只要算log次lowbit，所以只要log次询问就可以得出结果，询问区间和[L,R]，只要算两个前缀和，再相减即可。

```
int query(int x){
    int ans=0;
    while(x){
        ans+=c[x];
        x-=lowbit(x);
    }
    return ans;
}
//下面介绍更新操作
```

假如我们要让A[x]加上4，那么我们就需要在树状数组C中，让所有包含A[x] 的C[i]都加上4。

所以问题就变成了怎么找到包含A[x]的所有区间C[i]。

我们之前已经提过，C[i]掌管着[i-lowbit(i)+1,i]这个区间。但是lowbit不仅仅代表着掌管的区间大小，而且还是当前点距离父亲区间的距离。

从前页的图中，我们可以看到C[4]的lowbit值是4，所以它掌管长度为4的区间，同时，它距离父亲区间C[8]这个点的距离也是4。对于其他点也成立。(key)

所以我们要单点更新，只需要每次都要下标加上lowbit，然后更新节点信息即可。

```
//图可看下方
void add(int i,int k){
    while(i<=n){
        a[i]+=k;
        i+=lowbit(i);
    }
}

//接下来介绍树状数组的区间更新，单点查询的代码
```

Q个操作，询问某个点的值，或者给某个区间都加上一个值。

A是原数组，我们用D[i]表示差分数组，即D[i]=A[i]-A[i-1]。特别的，我们让D[1]=A[1]。

这样，我们再用树状数组C[i]去维护这个D[i]差分数组而不是A[i]原数组。我们要求A[i]就转变成了求D[1]到D[i]的前缀和，用树状数组实现。

现在我们如果要给某个区间[L,R]加上x，就转化成了单点更新，即让D[L]加上x，让D[R+1]减去x。

```
//提问为什么这里要用树状数组去维护这个差分数组  是为了降低时间复杂度 因为你正常的前缀和是o(n)的 但是我用了数组树状 就是o(log n)
int A[N],D[N],C[N];
int n;
int query1(int l,int r,int x){//[l,r]为区间 x为更新的值
    for (int i=1;i<=n;i++){
        D[i]=A[i]-A[i-1];
    }
    for (int i=1;i<=n;i++){
        add(i,D[i]); //就是上面那个操作 也就是维护树状数组
    }
```

```
    add(l,x);
    add(r+1,-x);
    return query(r)-query(l);
}
```

树状数组的3种功能代码

```
1.单点修改 和 区间查询
#include<bits/stdc++.h>
#define ll long long
using namespace std;
//数组数组学习
const int N=5e5+10;
int a[N];
int c[N];//这个c就是一个树状数组
//它的含义就是 c[x] 对应掌管 [x-lowbit(x)+1 ,x]
int n,m;
int lowbit(int x){
    return x&(-x);
}
//o(n) 的建树
void build(){
    for (int i=1;i<=n;i++){
        c[i]+=a[i];
        int j=i+lowbit(i);
        if(j<=n) c[j]+=c[i];
    }
}
//o(log(n)) 的建树
void add(int x,int k){
    while(x<=n){
        c[x]+=k;
        x=x+lowbit(x);
    }
}
void build1(){
    for (int i=1;i<=n;i++){
        add(i,a[i]);
    }
}

ll query(int x){
    ll ans=0;
    while(x>0){
        ans+=c[x];
//      cout<<c[x]<<' ';
        x=x-lowbit(x);
//      cout<<x<<endl;
    }
    return ans;
}
void print(){
    for (int i=1;i<=n;i++){
        cout<<c[i]<<' ';
    }
}
int main(){
```

```
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>a[i];
    build1();
    while(m--){
        int op,x,y;
        cin>>op>>x>>y;
        if(op==1){
            add(x,y);
//          print();
        }else {
//          cout<<query(y)-query(x-1)<<endl;
            cout<<query(y)-query(x-1)<<endl;
        }
    }
    return 0;
}
```

2.区间修改 和 单点查询

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=5e5+10;
int a[N],b[N],c[N];
int n,m;
int lowbit(int x){
    return x&(-x);
}
void build(){
    for (int i=1;i<=n;i++){
        c[i]+=b[i];
        int j=i+lowbit(i);
        if(j<=n) c[j]+=c[i];
    }
}
void add(int x,int k){
    while(x<=n){
        c[x]+=k;
        x+=lowbit(x);
    }
}
int ask(int x){
    ll ans=0;
    while(x){
        ans+=c[x];
        x-=lowbit(x);
    }
    return ans;
}
int main(){
    cin>>n>>m;
    for (int i=1;i<=n;i++) {
        cin>>a[i];
        b[i]=a[i]-a[i-1];
    }
    build();
    while(m--){
        int op;
        cin>>op;
        if(op==1){
```

```
            int x,y,k;
            cin>>x>>y>>k;
            add(x,k);
            add(y+1,-k);
        }else {
            int x;
            cin>>x;
            cout<<ask(x)<<endl;
        }
    }
    return 0;
}
```

3.区间修改 和 区间查询

```
//树状数组的 区间修改 和 区间查询
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e6+10;
ll a[N],b[N],t1[N],t2[N];
ll n,q;
void b1(){
    for (int i=1;i<=n;i++){
        t1[i]+=b[i];
        int j=i+(i&(-i));
        if(j<=n) t1[j]+=t1[i];
    }
}
void add1(ll x,ll k){
    while(x<=n){
        t1[x]+=k;
        x+=(x&(-x));
    }
}
ll ask1(ll x){
    ll ans=0;
    while(x){
        ans+=t1[x];
        x-=(x&(-x));
    }
    return ans;
}
void b2(){
    for (int i=1;i<=n;i++){
        t2[i]+=i*b[i];
        int j=i+(i&(-i));
        if(j<=n) t2[j]+=t2[i];
    }
}
void add2(ll x,ll k){
    while(x<=n){
        t2[x]+=k;
        x+=(x&(-x));
    }
}
ll ask2(ll x){
    ll ans=0;
    while(x){
        ans+=t2[x];
```

```
        x-=(x&(-x));
    }
    return ans;
}
int main(){
    cin>>n>>q;
    for (int i=1;i<=n;i++){
        cin>>a[i];
        b[i]=a[i]-a[i-1];
    }
    b1();b2();
    while(q--){
        ll op;cin>>op;
        if(op==1){
            ll l,r,x;cin>>l>>r>>x;
            add1(l,x);add1(r+1,-x);
            add2(l,l*x);add2(r+1,-(r+1)*x);
        }else {
            int l,r;cin>>l>>r;
            ll s1=1LL*(r+1)*ask1(r)-ask2(r);
            ll s2=l*ask1(l-1)-ask2(l-1);
            //cout<<s1<<' '<<s2<<endl;
            cout<<s1-s2<<endl;
        }
    }
    return 0;
```

## *线段树(需要用到二叉树相关的性质) （一定要开4倍空间）*

可以用来：O(logN) 询问区间和，O(logN)区间修改，单点修改。

线段树就是一棵二叉树，用节点代表一段区间。父节点的区间等于左右儿子节点的区间相加。例如父节点是[1,9]，那么左节点是[1,5],右节点是[6,9]。

对于一棵根节点的编号是1的二叉树，如果父节点的编号是i，那么左儿子节点的编号一定是2*i，右儿子节点的编号一定是2*i+1。所以可以用一个数组来模拟二叉树。
当然这里可以用位运算加速 所以我们一般是写成 p<<1 和 p<<1|1

我们让编号为1的节点掌管区间[1,n]，让他的左儿子掌管[1,mid]，右儿子掌管[mid+1,n]

递归下去，直到当前节点掌管的区间长度为1，即单个数字。

在回溯中，传递信息。

```
#include <bits/stdc++.h>
#define endl '\n'
#define int long long
// **线段树的打lazy 是同时更新节点的值的 这个点要注意**
using namespace std;

typedef long long ll;
```

```cpp
const int N=1e5+10;

int n,m;
int a[N];
int tr[N*4];
void build(int l,int r,int p){ //l 和 r 表示的是区间 p表示的是树上的节点的下标
    if(l==r){
        tr[p]=a[l];
        return ;
    }
    int mid = (l+r)/2;
    build(l,mid,p*2);
    build(mid+1,r,2*p+1);
    tr[p]=tr[2*p]+tr[2*p+1];
}
int lazy[N*4];
void push_down(int l,int r,int p){
    int mid = (l + r)/2;
    tr[2*p] += lazy[p] * (mid-l+1);
    tr[2*p+1] += lazy[p] * (r-mid);
    lazy[2*p]+=lazy[p];
    lazy[2*p+1]+=lazy[p];
    lazy[p]=0;

}
// l 和 r 为p这个节点所代表的区间 x 和 y 是我要修改的区间
void updata(int l,int r,int x,int y,int w,int p){
    //这里我们直接介绍区间修改的操作 因为单点修改 不就是区间长度为1吗
    //线段树之所以快 介绍快在 它的区间修改是采用了lazy标记这样一个东西
    if(x<=l&&y>=r){//对应完整的区间
        lazy[p]+=w; //打上lazy标记
        //同时更新这个节点的值
        tr[p]+=(r-l+1)*w;
        // lazy[p]=0;
        return ;
    }
    int mid = (l+r) / 2;
    //对于不是完整的区间lazy标记要下放 同时要删掉我这层（原有的）的标记
    push_down(l,r,p); //下传lazy标记 所以这里的参数的意思是说 把我这个区间往下
    if(x<=mid) updata(l,mid,x,y,w,2*p);
    if(y>mid) updata(mid+1,r,x,y,w,2*p+1);
    tr[p]=tr[2*p]+tr[2*p+1];
}
int query(int l,int r,int x,int y,int p){
    if(x<=l&&y>=r) return tr[p];
    int mid=(l+r)/2;
    int ans=0;
    push_down(l,r,p); //为什么在查询的时候也需要下放呢，这个你可以想一下
    if(x<=mid) ans+=query(l,mid,x,y,2*p);
    if(y>mid) ans+=query(mid+1,r,x,y,2*p+1);
    return ans;
}
signed main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    cin>>n>>m;
    for (int i=1;i<=n;i++) cin>>a[i];
    build(1,n,1);
    while(m--){
```

```
        int op,x,y;
        cin>>op>>x>>y;
        if(op==1){
            int k;
            cin>>k;
            updata(1,n,x,y,k,1);
        }else {
            cout<<query(1,n,x,y,1)<<endl;
        }
    }
    return 0;
}
```

## 线段树的运用

1）动态开点 （这里的线段树其实也是权值线段树了

```
//这题是用动态开点求 逆序对 所以没有建树的过程 其实是在modify里面了
//动态开点适用于什么题呢？ 要开很大的树 比如 1e9 然后但是询问只有1e5这样 同时这些个询问是不可以离线下来去离散化的时候 就可以用
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
const int N=5e5+10;
const int _=1e9+1;

int n;
int a[N];
struct ty{
    int sum[N*30];
    int ls[30*N],rs[30*N];
    int tot=1;

    void modify(int &p,int l,int r,int x){ //其实相当于build
        if(!p) p=++tot;
        if(l == r){
            sum[p] ++;
            return ;
        }
        int mid = (l+r)/2;
        if(x<=mid) modify(ls[p],l,mid,x);
        else modify(rs[p],mid+1,r,x);
        sum[p]=sum[ls[p]]+sum[rs[p]];
    }

    int que(int p,int l,int r,int x,int y){
        if(!p) return 0;
        if (x<=l && r<=y){
            return sum[p];
        }
        int mid =(l+r)/2;
        int ans=0;
        if(x<=mid) ans+=que(ls[p],l,mid,x,y);
        if(y>mid) ans+=que(rs[p],mid+1,r,x,y);
        return ans;
```

```
    }
}tr;
ll res;
int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    cin>>n;
    for (int i=1;i<=n;i++) cin>>a[i];
    for (int i=1;i<=n;i++){
        res+=tr.que(1,1,_,a[i]+1,_);
        int o=1;
        tr.modify(o,1,_,a[i]);
    }
    cout<<res<<endl;
    return 0;
}
```

2）多个lazy（easy版

能开多个lazy的是什么情况呢 是已知优先级的情况 比如 先乘再加

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
const int N=1e5+10;

ll n,q,mod;
ll a[N];

struct ty{
    ll tr[N*4],add[N*4],mul[N*4];

    void build(int l,int r,int p){
        add[p]=0;
        mul[p]=1;
        if(l==r){
            tr[p]=a[l];
            return ;
        }
        ll mid = (l+r)/2;
        build(l,mid,2*p);
        build(mid+1,r,2*p+1);
        tr[p]=(tr[2*p]+tr[2*p+1])%mod;
    }

    void push_down(int l,int r,int p){
        if(mul[p]!=1){
            tr[2*p] = (1ll*tr[2*p] * mul[p])%mod;
            tr[2*p+1]=(1ll*tr[2*p+1] * mul[p])%mod;
            mul[2*p] =( 1ll* mul[2*p] * mul[p])%mod;
            mul[2*p+1] = (1ll*mul[p] * mul[2*p+1]) %mod;
            add[2*p] = (1ll*add[2*p] * mul[p])%mod;
            add[2*p+1] = (1ll*add[2*p+1] * mul[p])%mod;
            mul[p]=1;
        }
        int mid=(l+r)/2;
```

```
        if(add[p]!=0){
            (tr[2*p]+=1ll*(mid-l+1)*add[p])%=mod;
            (tr[2*p+1]+=1ll*(r-mid)*add[p])%mod;
            (add[2*p]+=1ll*add[p])%=mod;
            (add[2*p+1]+=1ll*add[p])%=mod;
            add[p]=0;
        }
    }

    void Mul(int l,int r,int x,int y,int k,int p){
        if(x<=l&&r<=y){
            tr[p] = 1ll*tr[p]*k%mod;
            mul[p] = 1ll*mul[p]*k%mod;
            add[p] = 1ll*add[p]*k%mod;
            return ;
        }
        int mid=(l+r)/2;
        push_down(l,r,p);
        if(x<=mid) Mul(l,mid,x,y,k,p*2);
        if(y>mid) Mul(mid+1,r,x,y,k,2*p+1);
        tr[p]=(tr[2*p]+tr[2*p+1])%mod;
    }

    void Add(int l,int r,int x,int y,int k,int p){
        if(x<=l&&r<=y) {
            (add[p] += k)%mod;
            (tr[p] +=1ll* (r-l+1) * k )%=mod;
            return ;
        }
        int mid=(l+r)/2;
        push_down(l,r,p);
        if(x<=mid) Add(l,mid,x,y,k,2*p);
        if(y>mid) Add(mid+1,r,x,y,k,2*p+1);
        tr[p]=(tr[2*p]+tr[2*p+1])%mod;
    }

    int que(int l,int r,int x,int y,int p){
        if(x<=l&&r<=y){
            return tr[p]%mod;
        }
        int mid=(l+r)/2;
        push_down(l,r,p);
        int ans=0;
        if(x<=mid) ans+=que(l,mid,x,y,2*p)%mod;
        if(y>mid) ans+=que(mid+1,r,x,y,2*p+1)%mod;
        return ans%mod;
    }
}tr;

int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    cin>>n>>q>>mod;
    for (int i=1;i<=n;i++) cin>>a[i];
    tr.build(1,n,1);
    while(q--){
        int op,x,y,k;
        cin>>op>>x>>y;
        if(op==1){
```

```
            cin>>k;
            tr.Mul(1,n,x,y,k,1);
        }
        if(op==2){
            cin>>k;
            tr.Add(1,n,x,y,k,1);
        }
        if(op==3){
            cout<<tr.que(1,n,x,y,1)%mod<<endl;
        }
    }
    return 0;
}
```

### 3)01串的线段树

```
#include <bits/stdc++.h>
#define endl '\n'

using namespace std;

typedef long long ll;
const int N=1e4+10;
//这题比较棘手 因为分为树 和 树苗 所以我们要建两颗线段树
//一棵表示 活着的树 一棵表示 活着的树和树苗的数量
int l,n;
int a[N];
int res;
struct ty{
    int ans;//用来记录被砍掉的树
    int sum[N*4],lazy[N*4]; //lazy = 0 表示没操作 = 1 表示全部种上 = 2 表示全部砍掉

    void build(int p,int l,int r){
        if(l==r){
            sum[p]=a[l];
            return ;
        }
        int mid=(l+r)/2;
        build(2*p,l,mid);
        build(2*p+1,mid+1,r);
        sum[p]=sum[2*p]+sum[2*p+1];
    }

    void push_down(int p,int l,int r){
        if(lazy[p]==1){
            int mid =(l+r)/2;
            sum[2*p]=mid-l+1;
            sum[2*p+1]=r-mid;
            lazy[2*p]=1;
            lazy[2*p+1]=1;
            lazy[p]=0;
        }
        if(lazy[p]==2){
            int mid =(l+r)/2;
            sum[2*p]=0;
            sum[2*p+1]=0;
            lazy[2*p]=2;
```

```
            lazy[2*p+1]=2;
            lazy[p]=0;
        }
    }

    void zhong(int p,int l,int r,int x,int y){
        if(x<=l&&r<=y){
            lazy[p]=1;
            sum[p]=r-l+1; //这是是直接赋值 就好了
            return ;
        }
        push_down(p,l,r);
        int mid=(l+r)/2;
        if(x<=mid) zhong(2*p,l,mid,x,y);
        if(y>mid) zhong(2*p+1,mid+1,r,x,y);
        sum[p]=sum[2*p]+sum[2*p+1];
    }

    void cut(int p,int l,int r,int x,int y){
        if(x<=l&&r<=y){
            ans+=sum[p];
            lazy[p]=2;
            sum[p]=0;
            return ;
        }
        push_down(p,l,r);
        int mid=(l+r)/2;
        if(x<=mid) cut(2*p,l,mid,x,y);
        if(y>mid) cut(2*p+1,mid+1,r,x,y);
        sum[p]=sum[2*p]+sum[2*p+1];
    }

}tr[3];
int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    cin>>l>>n;
    for (int i=1;i<=l+1;i++){
        a[i]=1;
    }
    tr[1].build(1,1,l+1);
    tr[2].build(1,1,l+1);
    while(n--){
        int op,x,y;
        cin>>op>>x>>y;
        x++,y++;//因为题目是 0-l 我是 1-l+1
        if(op==0){//说明是砍 那树和树苗都要更新 1，2 都要更新
            tr[1].cut(1,1,l+1,x,y);
            tr[2].cut(1,1,l+1,x,y);
        }
        if(op==1){//因为是种 只种了树苗 所以只更新 2
            tr[2].zhong(1,1,l+1,x,y);
        }
    }
    //回答还剩多少课树苗
    cout<<tr[2].sum[1]-tr[1].sum[1]<<endl;
    //回答第二问
    cout<<tr[2].ans-tr[1].ans<<endl;
    return 0;
```

```
}
```

4)势能（均摊）线段树

```

```

# 分块

**分块**就是把一个东西分成多个块来进行维护操作，达到降低复杂度的一种操作，也被称为优雅的暴力。分块相对来说比较灵活，可以用在很多思路的优化上。

**具体操作：我们设数组长度为 N，分成 K 块，每块长度 N / K 。对于一次区间操作，对区间内部的整块进行整体的操作，对区间边缘的零散块单独暴力处理。所以，块数不能太少也不能太多。所以，一般来说，我们取得块长（块的长度）为 √ N**。这样我们的时间复杂度即为带** (N √ N) 的。这是一种根号算法。**

```
typedef long long ll;
const int N=1e5+10;
int n;
int belong[N];//用来记录每一个 属于哪一块
int block=sqrt(n);//一般是取根号n 当然有的时候也会取别的
void fk(){
    for (int i=1;i<=n;i++){
        belong[i]=(i-1)/block+1;
    }
}
//这个操作 有的时候是可以不用的 用了反而会T 因为你是可以直接算一下的 不用预处理
```

下面介绍区间修改的操作

之后的操作也是按照这三种情况分类

**给定一个长的为n数列，求出任意区间** [1，r] 的最大值(1<=l<=r<=n),当我们查询任意一个区间 [1，r] 时，如果 l 所在的块与 r 所在的块相同，如 [1，2]，则直接暴力查询即可，时间复杂度 O(√N)

**若其不在一个块但是块是相邻的，一样是暴力查询，时间复杂度 O(√N)**

**若其块不相邻，如 [1，10]，我们先处理两边的边块角块，先暴力查询 1 和 10 所在的块内最大值，最后直接查询中间块内最大值即可，时间复杂度 O(√N)****

```
//区间修改 区间查询
int val[N];
int tag[N];
void query(int l,int r,int c){//c为修改的值
    for (int i=l;i<=min(belong[i]*block,r);i++){
        val[i]+=c;
    }//这是对于左边角料的暴力枚举
    if(belong[l]!=belong[r]){
        for (int i=(belong[r]-1)*block+1;i<=r;i++){
            val[i]+=c;
        }
    }//这是对于右边角料的暴力枚举
```

```
    for (int i=belong[l]+1;i<=belong[r]-1;i++){
        tag[i]+=c;
    }//这是对于中间的完整的区间的 这里的操作有点像线段树的懒惰标记
}
```

至于最后一步打上 tag 主要是为了减少时间复杂度 那它该怎么用呢 比如我要查询一下[l,r]的区间最大值 对于这些完整的块 里面就会有一个最大值 然后再让它加上这个tag就好了

//区间查询小于 X 的元素个数

利用上面的思想，可以先处理出 Tag 数组，并对于每个区间维护一个 Vector ，然后用 Lower_bound 直接暴力求解即可 当然这里的vector里面是要排好序的

然后对于不是完整的就暴力枚举就好了 就a[2]+tag[i] 看是否小于x

而对于完整的块 就x-tag[i] 然后再去lower_bound 这样时间复杂度就降下来了

//查询 X 的前驱 前驱是指比x小的数里面的最大值

将块内查询的二分进行修改，由于是要找的是 X 的前驱，因此暴力找左右不完整块的比 X 小最大值，再二分找整块比 X 小的最大值，取其中最大即可

其实与上面操作相差不大 只是再多几步罢了

//区间开方 除法

注意到开方这个操作并不会进行多次，我们可以限定区间执行次数之后暴力实现即可

那如何去限定这个次数呢 比如我们让这个区间的最大值变成1了 就可以停止了

//区间修改和区间查询众数 区间修改和上面一样

首先我们把这个数组分为 k 块来看待，然而每次查询的 L - R 则有几种可能，

当 L 与 R 属于同一个区间时，我们只需要暴力计算即可

当 L 与 R 在相邻区间时，同样也可以暴力计算

当 L 与 R 在不相邻的两块区间中，我们需要维护一个数组 Min[ I ][ j ] 表示为第 I 块到第 j 块出现次数最多且最小的数，这样我们暴力查询 L 和 R 所在块后可以利用 Min 数组进行答案的计算

下面着重介绍第三种情况的处理 它其实是一种预处理的思想 我先去把min这个数组处理好 等你来查询的时候 遇到第三种就直接调用出来 从而达到降低时间复杂度的思想

```
//区间查询众数的第三种情况的预处理
int a[N];//即原数组
int cnt[N];
int MIN[N][N];
void query(){
    for (int i=1;i<=belong[n];i++){//遍历每一个块
        int ans=0,res=0;//ans 记录当前出现次数最多且最小的数 res记录的当前出现次数最多的次数
        for (int j=(i-1)*block+1;j<=n;j++){//从第i块的起点 到最后为止
            cnt[a[j]]++;
            if(cnt[a[j]]>ans||cnt[a[j]]==ans&&a[j]<ans){//这里就是找到最小的那个众数
                res=cnt[a[j]];
                ans=a[j];
            }
            MIN[i][belong[j]]=ans; //当前i块到j所在的块的当前答案
        }
        memset(cnt,0,sizeof(cnt));//每一次都要清空重来一遍
    }
}
```

这里再来介绍一种好写的板子 例题

上面介绍的是用 从 1到n的写法

我们这种是从 0-(n-1)

所以我们在计算它属于哪一块时 就很 简单 直接 i/block

然后其他和上面思路基本一致

这题就是用分块维护 区间 最大值的这么一道题

```
#include <bits/stdc++.h>
#define int long long
#define endl '\n'
using namespace std;
const int N=2e5+10;
```

```
int _;
int n,t,q;
int A[N];
int tag[N];
int block;
void fk(){

}
void query(int l,int r){
    int res=0;
    l--,r--;
    int x=l/block;
    int y=r/block;
    //这种是处理相邻 or 在同一区间
    //直接暴力for
    if(x==y||x+1==y){
        for (int i=l;i<=r;i++){
            res=max(res,A[i]);
        }
    }else {
        //这个是不相邻的 （其中可以把上面并下来写
        for (int i=l;i<(x+1)*block;i++){//左边角料
            res=max(res,A[i]);
        }
        for (int i=x+1;i<y;i++){//中间完整的
            res=max(res,tag[i]);
        }
        for (int i=y*block;i<=r;i++){//右边角料
            res=max(res,A[i]);
        }
    }
    cout<<res<<' ';
    if(res>=q) cout<<"YES"<<endl;
    else cout<<"NO"<<endl;
}
void solve(){
    cin>>n>>t>>q;
    block=sqrt(n);
    for (int i=0;i<n;i++){
        cin>>A[i];
        int bl=i/block;
        if(A[i]>tag[bl]) tag[bl]=A[i];
    }
    int L,R;
    while(t--){
        cin>>L>>R;
        query(L,R);
    }
}
```

# 莫队 (是对询问的分块)

```
//莫队是一种在分块的前提下在去做操作的一种数据结构 但是这里的分块不是按照根号n来的
int n,m;//n为总长度 m为询问的条数
struct xunwen{
    int l,r,id;//这里的id是它本来的顺序 因为我后面排序了
}querys[N];//用来记录每一个询问
int ans[N];
int res;//用来记录答案
bool cmp(xunwen a,xunwen b){
    //... 对于区间 [l,r], 以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序
}
void move(int pos,int sign){
    //...  这里是实现o(1)转移的 不同的题目不一样
}
void solve(){
    int curL=1;//这里的 l和r是指当前的l r然后初始是 1和0当然别的可能也可以 但是不可以是 0 0不然就是也看成是一个区间从而造成影响
    int curR=0;
    int block=ceil(sqrt(n));//最推荐 开 根号m  虽然这样好像也行(oiwiki上这样写)
    sort(querys,querys+n,cmp); //这里的排序是按照每个询问的左边所属于的块的编号从小到大 如果相同 则按右边界的真实的大小 从小到大
    for (int i=0;i<m;i++){
        auto q=querys[i];
        //接下来就是4个循环 从当前区间 然后o(1)转移
        //然后为什么有的是先加 有的是后加 因为有的操作时加上这一格的贡献 有的则是减去这一格的贡献
        while(curL>q.l) move(--curL,1);//所以这里的1 就是代表我是要加上的
        while(curR<q.r) move(++curR,1);
        while(curL<q.l) move(curL++,-1);//这里的-1 就是代表我要减去
        while(curR>q.r) move(curR--,-1);
        ans[q.id]=res;//你经过这4个移动你就可以知道答案了 也就可以存下来的 之后再按照题目原本问的顺序输出即可
    }
}
//这些就是莫队主函数的总体框架 主要会变化的就是这个move函数 你不同的题目肯定不一样 这个就是看你自己做的题和自己的理解了


板子
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
const int N=1e6+10;
ll n,q;
ll sq; //是根号n哦
ll a[N];

struct node{
    ll l,r,id;
}Q[N];

bool cmp(node a,node b) { //这个是莫队的精髓 它的排序是按 奇数从小到大 偶数从大到小
    ll aa=a.l/sq;  ll bb=b.l/sq;
    //对于不同块的 按块从小到大排
    if(aa!=bb) return aa<bb;
```

```
    //对于同一块 按照上面的排序规则
    //但是为什么只用 管 r 呢 ？
    /*
        因为按左端点排序 还是会造成左右（来回）摆动的幅度还是较大
        而按右端点则可以减少摆动幅度，从而减小时间复杂度
    */
    if(aa%2==0) return a.r>b.r;
    return a.r<b.r;
}
ll cnt[N]; //记录每个颜色出现的次数
ll result,ans[N]; //记录答案  因为莫队是一个离线的操作 所以别忘了记录答案哦

void change(int color,int delta){ //一个是高数是什么颜色 一个是告诉是该加还是减
    int prev = cnt[color];
    cnt[color] += delta;
    result += cnt[color]/2-prev/2;
}
int main(){
    ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    cin>>n;
    sq = sqrt(n);
    for (int i=1;i<=n;i++) cin>>a[i];

    cin>>q;

    for (int i=1;i<=q;i++){
        cin>>Q[i].l>>Q[i].r;
        Q[i].id=i;
    }

    ll L=1,R=0;  //最开始先设置成空区间 这样不会对答案有影响
    sort(Q+1,Q+1+q,cmp);
    for (int i=1;i<=q;i++){
        ll L2=Q[i].l;
        ll R2=Q[i].r;
        //记住先扩再缩
        while(L>L2){
            L--;//因为是扩 所以这个答案我是考虑过的 所以先减再change答案
            change(a[L],1);
        }
        while(R<R2){
            R++;
            change(a[R],1);
        }
        while(L<L2){
            change(a[L],-1);
            L++; //因为是缩 所以先改变再减 不然我这个点就没去改变了
        }
        while(R>R2){
            change(a[R],-1);
            R--;
        }
        //别忘了记录答案
        ans[Q[i].id]=result;
    }
    for (int i=1;i<=q;i++){
        cout<<ans[i]<<endl;
    }
```

```
    return 0;
}
```

# *倍增算法(log n)*

基本用法 为了查找单调数组中的某一数值

```
eg: 在{2,5,7,11,19} 中查找最大的小于12的数字
    朴素 从头开始一个一个遍历找到
    二分
    倍增：先设定一个增长长度 p 和已经确定的长度l，现在要确定a[l+p]是否满足条件 如满足 p>>=1;
    不满足 则缩小范围 p<<=1;
l=0;
p=1;
while(p)//如果还能扩增范围(l)就继续
{
    if(a[l+p]<12)
    {
        l+=p;//增加已知范围
        p<<=1//成倍增长，相当于p*=2
    }
    else
    {
        p>>=1;//缩小范围
    }
}
cout<<a[l];
```

## 运用：用倍增求循环节

```
初始状态为 1  2  3  ...  n
cnt[N] 数组中计的是每个数的循环节（即周期）
for (int i=1;i<=n;i++){
        cnt[i]=1;
        while(s[b[i]]!=i){
            b[i]=s[b[i]];
            cnt[i]++;
        }
    }
```

# *ST表*

是对于 求解 多次询问 区间的 gcd max min的一种数据结构

对应的 它也就能 维护一个区间的 gcd max min

例题1

这是一道 st表 + 二分的板子题　　我们借题目来讲思路

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=2e5+10;
const int mod=998244353;
int n;
int a[N];
int lg[N];
int st[N][20]; //st[i][j] 存的是 存的 从i到 i+2^(j)-1 的gcd
int que(int l,int r){
    int k=lg[r-l+1];
    return __gcd(st[l][k],st[r-(1<<k)+1][k]);
    //这里是 我去查询l 到 r 区间的gcd 但是 r-l+1 的长度不一定是 2的倍数
    //所以我们 我们采用 从 l 到 l+2^k-1   然后  r 到 r前面 长度 也是 2^k的
    //这样虽然中间有重复 但是重复一点点有 什么所谓呢
}
ll ksm(ll a,ll b){//为了求逆元用的
    ll ans=1;
    a = a % mod;
    while(b){
        if(b&1) ans = ans * a % mod;
        b>>=1;
        a = a * a % mod;
    }
    return ans;
}
int main(){
    lg[0]=-1;//log函数 算log 2 n为几
    for (int i=1;i<N;i++) lg[i]=lg[i>>1]+1;
    cin>>n;
    for (int i=1;i<=n;i++) cin>>a[i];
    for (int i=1;i<=n;i++) st[i][0]=a[i]; //最开始所有的数的gcd都是它自己
    for (int j=1;j<=19;j++){
        for (int i=1;i+(1<<(j-1))<=n;i++){
            //这里是维护所以区间的gcd
            st[i][j]=__gcd(st[i][j-1],st[i+(1<<(j-1))][j-1]);
        }
    }
    int Ans=0;
    for (int i=1;i<=n;i++){
        int R=i;
        while(R<=n){
            int l=R,r=n,mid;
            while(l<=r){
                mid=l+r>>1;
                if(que(i,R)==que(i,mid)) l=mid+1;
                else r=mid-1;
            }
            //这里为什么可以这样呢 是因为 我这样找完之后 长度 为 3 4 5 6 的gcd都是 a
```

```
            //然后我要求这些聚会的合 然后我每一次 聚会结束的快乐度的和 是所有人的和
            //所以我这4个区间的所有的快乐度的和 不就是（3+4+5+6）*a 吗
            //然后 3 4 5 6 不是一个等差数列吗
            //那所以这个和 不就可以 首项 加 末项 ×项数 /2 吗
            Ans = (Ans+ 1ll*(r-R+1)*(R+r-2*i+2)/2%mod*que(i,R)%mod)%mod;
            R=r+1;
        }
    }
    ll inv=1ll*n*(n+1)/2;//求逆元 这样就不用除法了
    inv=ksm(inv,mod-2);
    cout<<Ans*inv%mod<<endl;
    return 0;
}
```

# 前缀和和差分

前缀和数组 和 差分 数组 是为了多次 解决区间修改 和 区间询问的这么一种数据结构
这里先简单讲一下你老是会错的点

### 差分例题

```
前缀和 的 关键
qzh[i]=a[i]+qzh[i-1];
差分 的关键
pre[i]=a[i]-a[i-1];
至于为什么是这样 因为 他们之间满足 差分数组前缀和一下就会变成原数组 原数组前缀和一下就会变成前缀和数组 也就是 前缀和和差分 是互逆的
```

下面介绍 二维的前缀和 和 差分

# 二叉树

这里这是做粗略的讲解 这是 让你稍作了解用的

这个代码主要是解决 给 中序 加任意一个序的遍历 求出这棵树 （当然也可以层序输出

### 例题

```
#include <bits/stdc++.h>
using namespace std;
int n;
queue <int> q;
int mid[50],suf[50]; //存中序 和 后序
int l[50],r[50],root;//左子树 和 右子树 根节点
//5个参数 分别什么意思呢 第一个 是 我要 建的这个树的根
//去后序寻找的两个边界值    去中序寻找的两个边界值
void dfs(int &p,int sufl,int sufr,int midl,int midr){
    p=suf[sufr];
    int pos=0;//用来表示 我这个根在 中序 中的位置
    for (int i=midl;i<= midr;i++){
        if(mid[i]==p) {
```

```
            pos=i;
            break;
        }
    }
    int poslen=sufl+pos-midl-1;//用来 标记我待会要建树的范围
    //如果左子树 还没建完 则去找
    if (pos!=midl) dfs(l[p],sufl,poslen,midl,pos-1);
    //同理
    if (pos!=midr) dfs(r[p],poslen+1,sufr-1,pos+1,midr);
}
int main(){
    cin>>n;
    for (int i=1 ;i <= n;i++) cin >> suf[i];
    for (int i=1 ;i <= n;i++) cin >> mid[i];
    dfs(root,1,n,1,n);
    //接下来是 广搜 输出层序遍历
    q.push(root);
    while(q.size()){
        int x=q.front();
        q.pop();
        cout<<x<<' ';
        if(l[x]) q.push(l[x]);
        if(r[x]) q.push(r[x]);
    }
    return 0;
}
```

这是给先序和中序然后输出后序

(数据结构课 一定要按这样写 包括变量名啊 啥的)

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=2e5+10;
char inod[30];
char preod[30];
typedef struct BiTNode{
    char data;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;
void PreInOd(char preod[],int i,int j,char inod[],int k,int h,BiTree &T){
    T = (BiTree) malloc(sizeof(BiTNode));
    T->data=preod[i];
    int m=-1;
    //去中序中寻找 我的根节点的位置
    for (int p=k;p<=h;p++){
        if (inod[p]==T->data){
            m=p;
            break;
        }
    }
    if(m==k) {
        T->lchild=NULL;
    }else {
        PreInOd(preod,i+1,i+m-k,inod,k,m-1,T->lchild);
    }
    if(m==h) {
```

```
        T->rchild=NULL;
    }else {
        PreInOd(preod,i+m-k+1,j,inod,m+1,h,T->rchild);
    }
}
void PostOrderTraverse(BiTree &T){
    if(T->lchild!=NULL){
        PostOrderTraverse(T->lchild);
    }//不可以写成 else if的结构 因为你这样 会有一半的树被你删掉
    if (T->rchild!=NULL){
        PostOrderTraverse(T->rchild);
    }
    cout<<T->data;
}
int main(){
    BiTree T;
    scanf("%s",preod);
    scanf("%s",inod);
    int n=strlen(preod);
    PreInOd(preod,0,n-1,inod,0,n-1,T);
    PostOrderTraverse(T);
    return 0;
}
```

下面是二叉树的建树 这里的建树方法是给你一个序列 一般是默认第一个元素就是根节点 然后我们这里是按 二叉搜索树的模式去建 让我的左边的元素小于等于我 让我右边的元素大于我

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
struct node{
    int va;//节点的值  但是我们对于每个节点还是用它在数组的中的下标来存 不然 你数组就可能会存不下
    int l,r;//左子树 右子树 的根节点
    int deg;//我这个节点的深度
}tree[1010];
int n;
int a[1010];
int main(){
    cin>>n;
    for (int i=1;i<=n;i++) cin>>a[i];
    tree[1].va=a[1];
    tree[1].deg=1;
    for (int i=2;i<=n;i++){
        int root=1;
        while(1){
            if(a[i]<=tree[root].va){ //往左子树去找
                if(tree[root].l==0){ //找到了
                    tree[root].l=i;
                    tree[i].va=a[i];
                    tree[i].deg=tree[root].deg+1;
                    break;
                }
                root=tree[root].l; //没找到 接着往下找
            }else {
                if(tree[root].r==0){
                    tree[root].r=i;
                    tree[i].va=a[i];
```

```
                    tree[i].deg=tree[root].deg+1;
                    break;
                }
                root=tree[root].r;
            }
        }
    }
    int mdeg=0;
    int ans=0;
    for (int i=1;i<=n;i++){
        if(tree[i].deg>mdeg) mdeg=tree[i].deg;
    }
    for (int i=1;i<=n;i++){
        if(tree[i].deg==mdeg||tree[i].deg==mdeg-1){
            ans++;
        }
    }
    cout<<ans<<endl;
    return 0;
}
```

# 队列

最简单的约瑟夫环问题 就是用队列去模拟 当数到要出列的时候就真出列 其他时候就是从队头出队 然后 立马 入队

链接

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e5+10;
queue <int> q;
int n,m;
int main(){
    cin>>n>>m;
    for (int i=1;i<=n;i++) q.push(i);
    int cnt=1;
    while(q.size()){
        if(cnt==m){
            cout<<q.front()<<' ';
            q.pop();
            cnt=1;
        }else {
            cnt++;
            q.push(q.front());
            q.pop();
        }
    }
    return 0;
}
```

# 最短路算法

（注意 边中出现负数 F B 可以用　出现负环图 只能 用 B　其他情况 F D B 都可以 根据题目要求选择）　B就是SPFA

## *Floyd算法*

### 例题

```
//本质是一种dp
//适用于求任意结点之间的最短路 适用于任何图，无论是否有向或者边权正负（不能有负环）
//0(n3) 得用邻接矩阵存图
定义状态 f[k][x][y]表示 只经过节点1-k，节点x到节点y的最短长度(即 从x走到y 期间只能选择 1-k的节点 所产生的最短长度)
    边界条件 f[0][x][y]= 0,边权,正无穷
    x=y:0   x,y连通:边权  x,y不连通:正无穷
状态转移方程 f[k][x][y]=min(f[k-1][x][y],f[k-1][x][k]+f[k-1][k][y]);
//如何理解？ 即每来一个点 我来比较我走这个点和不走这个点 那个距离短
代码实现(k x y 不可换顺序)
for (int k=1;k<=n;k++){
    for (int x=1;x<=n;x++){
        for (int y=1;y<=n;y++){
            f[k][x][y]=min(f[k-1][x][y],f[k-1][x][k]+f[k-1][k][y]);
        }
    }
}
//优化 观察状态转移方程 发现 从k-1到k 都只是从自己这一维转移过去 所以可以优化掉
for (int k=1;k<=n;k++){
    for (int x=1;x<=n;x++){
        for (int y=1;y<=n;y++){
            f[x][y]=min(f[x][y],f[x][k]+f[k][y]);
        }
    }
}
```

### 例题2　这是区间max

```
#include <bits/stdc++.h>
using namespace std;
const int N=2e5+10;
int A[N];
int st[N][20];//可以用来 维护 区间 max min gcd
int lg[N];
int n,t,q;
void que(int l,int r){
    int k=lg[r-l+1];
    int ans=max(st[l][k],st[r-(1<<k)+1][k]);
    if (ans>=q)  cout<<ans<<' '<<"YES"<<endl;
    else cout<<ans<<' '<<"NO"<<endl;
}
int main(){
    lg[0]=-1;
    for (int i=1;i<=N;i++) lg[i]=lg[i>>1]+1;
```

```
    cin>>n>>t>>q;
    for (int i=1;i<=n;i++) cin>>A[i];
    for (int i=1;i<=n;i++) st[i][0]=A[i];
    for (int j=1;j<=19;j++){
        for (int i=1;i+(1<<(j-1))<=n;i++){
            st[i][j]=max(st[i][j-1],st[i+(1<<(j-1))][j-1]);
        }
    }
    while(t--){
        int L,R;
        cin>>L>>R;
        que(L,R);
    }
    return 0;
}
```

# *SPFA算法*

在讲这个之前先引入一下Bellman-Ford算法中的松弛操作

首先我们定义一个数组dis[x]表示从初始点到x点的 **估计** 最短路长度（可能是中间状态）。

//松弛操作 *(key)对于边(x, y)我们存在dis[y] = min(dis[y], dis[x] + w(x, y))成立。

这样做的含义是显然的：我们尝试用S->x这个已知的最短路来更新S->y的最短路长度，这条路径如果优于初始的估计最短路长度就可以进行更新。

在最短路存在的时候，最长的最短路长度可以经过n-1条边，于是可以得出：如果进行了n轮以上循环就能证明途中存在负环且S点在负环上。（key）

需要注意的是，Bellman-Ford如果没有出现负环的情况也不一定说明图中一定没有负环，只能证明S点不在一个负环上。如果只想判断途中是否存在负环，可以建立一个超级源点连接所有的点边权为0。（所以我们对于Bellman-Ford只是引入）

```
//基于Bellman-Ford算法我们发现其实很多时候不需要这么多次松弛操作，只有前面被松弛操作过的点所连的边才需要继续进行松弛。
//那么我们用队列来维护【哪些结点可能会引起松弛操作】，就能访问边了。
//SPFA也可以用于判断s点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少n条边时，说明s点可以抵达一个负环。
所以其思路也就是 我先给定一个起点 s点 然后放入队列中 去遍历队列的头的节点的所有出边 做松弛操作 然后出队
    然后做过松弛操作的新的点(即判断过的 就不去判断了 说明要开一个vis数组) 再入队 再重复 直到队空 同时还需要开cnt数组 记录我最短路经过了多少条边 用来判断 s点是否在一个负环内
代码实现
const int N=3e5+50;
int dis[N];
int vis[N];
int cnt[N];
queue <int>q;
vector<pair<int,int>>adj[N];
int n;
void add_edge(int u,int v,int w){
    adj[u].push_back({v,w});
}
```

```
bool SPFA(int st){//给定一个起点
    memset(dis,0x3f3f3f3f,sizeof(dis));//先对距离赋值 为正无穷 默认全都达不到
    memset(vis,0,sizeof(vis)); //用来记录是否在队列中
    memset(cnt,0,sizeof(cnt));//用来记录做了几次松弛操作 用于之后判断是否有负环
    dis[st]=0;
    vis[st]=1;
    q.push(st);
    while(q.size()){
        int x=q.front();
        q.pop();
        vis[x]=0;
        for (auto [y,v]:adj[x]){
            if(dis[y]>dis[x]+v){
                dis[y]=dis[x]+v;//做松弛操作
                if(!vis[y]){//为什么一定要有这个vis数组 看下面详解
                    if(++cnt[y]>=n) return 1;//说明有负环
                    vis[y]=1;
                    q.push(y);
                }
            }
        }
    }
    return 0;
}
```

为什么要有这个 vis数组呢 而且为什么入队就要置1 出队就要置0

```
    //因为这里的vis数组是用来记录你是否在队列中的 是用来判重的 并不是什么是否访问过 （key）
    //为什么需要判重呢 因为一个点在队列中出现多次是没有意义的 因为你只要出现过我就要对它松弛 你来两次 和我在一次是无所谓的 你反而多次来 还会增加时间复杂度
```

因为一个点可能被多次访问 但是我实际只会做一次松弛操作 所以如果 这个点已经在队列里了 那尽管我后面虽然会去更换松弛操作的指 但是依旧不影响我去记录它只会被松弛一次 (但是一个点不能入队多次 指在我一次循环中 即在一次遍历出边时)

# *Dijkstra算法*

Dijkstra算法是一种求解非负权图上单源(单个起点)最短路的算法

将所有的点分为两个集合，一个是已经确定最短路长度的点集称为S，一个是未确定最短路长度的点集称为T。
一开始所有的点都在T集合中并且除了dis[s]=0（s为初始点）以外所有点的dis为+∞//dis数组就是用来记最短路为多少
//这个T集合我们就用优先队列来维护 去了S集合 就相当于出队列
每次在T中选取一个最短路长度最小的点，移到S中，对于新加入S的那个点的所有出边进行松弛操作。

直到T集合变为空集合停止操作。

在算法竞赛中，我们一般使用优先队列来维护最小最短路的那个点。//stl里有用法
时间复杂度为 O((n+m)log n)
代码实现

```
const int N=2e5+50;
int n;
int dis[N],vis[N];
vector<pair<int,int>>adj[N];
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair
//一个注意点 这里的优先队列 如果用了 pair 也是分第一关键字和第二关键字的
//所以你一定要把 dis放前面
<int,int>> > q;
```

```
void add_edge(int u,int v,int w){
    adj[u].push_back({v,w});
}
void Dijkstra(int st){
    memset(dis,0x3f,sizeof(dis));
    memset(vis,0,sizeof(vis));
    //dis[st]=0; 这里写不写 都行
    q.push({dis[st]=0,st}); //优先队列中 第一个表示 最短距离(dis) 第二个表示这条边的起点
    while(q.size()){
        int x=q.top().second;
        q.pop();
        if(vis[x]) continue;
        vis[x]=1;//为什么这里是这里才把它vis置1
        for (auto [y,v]:adj[x]){
            if(dis[y]>dis[x]+v){
                dis[y]=dis[x]+v;
                q.push({dis[y],y});//这里的疑问 也是同上面那个原因
            }
        }
    }
}
//原因：因为这里的vis和SPFA表示的含义不一样 同时这里不需要再置0这种操作 因为在这种算法下 最短路只会被选一次
```

## *DP版Dijkstra*

```
const int N=2e5+50;
int n;
int dis[N],vis[N];
int f[N];//用来记这个方案数
vector<pair<int,int>>adj[N];
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>> > q;
void add_edge(int u,int v,int w){
    adj[u].push_back({v,w});
}
void Dijkstra(int st){
    memset(dis,0x3d,sizeof(dis));
    memset(vis,0,sizeof(vis));
    dis[st]=0;
    f[st]=1;//刚开始的方案数肯定为1嘛 这个好理解
    q.push({dis[st]=0,st});
    while(q.size()){
        int x=q.top().second;
        q.pop();
        if(vis[x]) continue;
        vis[x]=1;
        for (auto [y,v]:adj[x]){
            if(dis[y] > dis[x] + v) {
                dis[y] = dis[x] + v;
                f[y] = f[x];//如果我这里要松弛操作 那我的方案数就是不变的 就从你转移过来就好了
                q.push({dis[y], y});
            } else if(dis[y] == dis[x] + v) {
                f[y] += f[x];//但是如果我不用松弛操作 就我现在的最短路长度和你一样 那我的方案数就要累计了
```

```
            }
        }
    }

}
```

# 最小生成树

## *Kruskal算法*

```
//每次取最短的一条边判断是否成环，若不成环就为最小生成树上的。
//把每个点一开始当成一个一个集合，一条边代表将两个集合合并，如果两个点有同样的祖先代表成环。(并查集去维护)
首先 n个节点的树 一定有n-1条边
那我们是用什么来存这个最小生成树呢 是用并查集(也可以说 这棵最小生成树看起来是一个并查集)
思路 把那些能构成树的节点放到并查集，然后每次选出一条最短的边之后 去看这条边的两边的两个端点是否是同一个root(即祖先) 也即判断它是否成环了
代码实现
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=2e5+4;
int n,m;
int cnt;//用来记 有这颗树有几条边了 如果==n-1 说明可以生成最小树 如果不等于 则不可以
int ans;//用来记录这颗最小生成树为多少(即边权和)
struct edge{//用到了直接存图法 是存边的
    int x,y,val;
}adj[N];
int f[N];//用来记每个节点的祖先
int Get_root(int x){//找root 找祖先的这么一个函数
    if(f[x]==x) return x;
    return f[x]=Get_root(f[x]);
}
int main(){
    cin>>n>>m;
    for (int i=1;i<=n;i++) f[i]=i;
    for (int i=1;i<=m;i++){
        cin>>adj[i].x>>adj[i].y>>adj[i].val;
    }
    sort(adj + 1, adj + m + 1, [&](const edge &p, const edge &q) {
        return p.val < q.val;
    }) ;//降adj按边从小到大排 因为我之后每一次都是取出最小的边
    for (int i=1;i<=m;i++){
        int x=adj[i].x,y=adj[y].y;
        if(Get_root(x)==Get_root(y)) continue; //判断是否成环了 然后是就跳过就好了
        f[x]=y;
        ans+=adj[i].val;
        cnt++;
    }
    if(cnt==n-1) cout<<ans<<endl;
    else cout<<"no"<<endl;
    return 0;
}
```

# 图论

## *拓扑排序*

```
//只能用在DAG(有向无环图)
//应用
/*在一个 DAG（有向无环图） 中，我们将图中的顶点以线性方式进行排序，使得对于任何的顶点 u 到 v 的有向边(u,v)，都可以有 u 在 v 的前面。
//拓扑排序的目标是将所有节点排序，使得排在前面的节点不能依赖于排在后面的节点。
常见于DAG上做dp*/
//借助队列实现的
const int N=1e5;
vector <int>G[N];
queue <int> q;
int u,v,n,m;
int deg[n];
//思想是 先找一个 入度为0的顶点 然后再把它相邻的度减一 再重复(集合队列)
void toposort(){
    for (int i=0;i<m;i++){
        cin>>u>>v;
        G[u].push_back(v);//加边
        deg[v]++;//记的是入度
    }//用邻链表存图
    for (int i=1;i<=n;i++){
        if(!deg[i]){//先找的一个度为0的顶点
            q.push(i);
        }
    }// *自己直接记错了*
    while(!q.empty()){
        int u=q.front();
        q.pop();
        id[u]=++idx;//用来存 排好序
        for (auto v:G[u]){
            deg[v]--; //相邻的 度减一（相当于删除了u这个顶点）
            if(!deg[v]){
                q.push(v);//重读上述过程
            }
        }
    }
}
```

## *Tarjan*

是用来找到 一个无向图里的 所有强连通分量

一个孤立的点 也算是 强连通分量 板子题

学习资料

```
思路 我们用先遍历 再输出点的 dfs方式 去遍历 一遍图
然后对于每个点 我们维护两个量 一个 i 表示 dfs中 x点最初被访问的时间点
                              j 表示 x通过无向边，可以回溯到的最早的时间点
关于时间点的定义 time=1  第一次访问到你 就是 1 然后我又去访问 别人了 那别人 就是2
```

同时 在 dfs中 遇到一个点 就入栈 然后再按dfs来
然后最后到 dfs该输出点的时候 看 这个点 是否在 栈中 以及是否被dfs里 如果没被 dfs就先dfs它 （具体看代码 因为这种可能就是正常的dfs的过程）
dfs回来到 x的时候 就更新 它的j （如果可以的话）
当dfs完它的出边后
去看它的i j是否相等 如果相等 则出栈到 x为止
这样 从栈顶到x的这一整块 就是一个强连通分量了

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=5e4+10;
vector <int> G[N];//邻链表存图
int n,m;
stack <int> st; //栈
int te=1; //时间
int dfn[N],low[N],flag[N];
//dfn 维护 i  low 维护 j   flag 维护 它是否还在栈中
int res;
void dfs(int x){
    st.push(x);
    flag[x]=1;
    dfn[x]=te++;
    low[x]=dfn[x];
    for (auto v:G[x]){//遍历出点（边）
        if (dfn[v]==0){
            dfs(v); //正常dfs
            low[x]=min(low[x],low[v]); //更新时间戳
        }else if (flag[v]==1){
            low[x]=min(low[x],low[v]);
        }
    }
    if(dfn[x]==low[x]){ //出栈
        int cnt=0;
        while (st.top()!=x){
            flag[st.top()]=0;
            st.pop();
            cnt++;
        }
        flag[st.top()]=0;
        st.pop();
        cnt++;
        if(cnt>1) res++;
    }
}
int main(){
    cin>>n>>m;
    for (int i=1;i<=m;i++){
        int u,v;
        cin>>u>>v;
        G[u].push_back(v);
    }
    for (int i=1;i<=n;i++){
        if(dfn[i]==0){
            dfs(i);
        }
    }
    cout<<res<<endl;
    return 0;
```

```
}
```

# 数学

## *gcd和lcm的性质*

```
gcd(a,a)=gcd(a,a)
gcd(a,b)=gcd(a,a-b) a>b;
gcd(a,b)=gcd(b,a-b) a>b;
```

## *错排*

```
dp[1]=0;
dp[2]=1;
dp[3]=2;
dp[n]=(n-1)*(dp[n-1]+dp[n-2]);
```

## *逆元*

```
a/b%c=ab^(-1)%c
```

同时根据费马小定理

取b^(-1)=b^(c-2)即可 (其中c为素数)

所以 a的逆元 为 a^(p-2) p为质数

# 快速幂

```
typedef long long ll;
const ll mod=1e5+10;
ll ksm(ll a,ll b){
    ll ans=1;
    while(b){
        if(b&1) ans=ans*a%mod;
        a=a*a%mod;
        b>>=1;
    }
    return ans;
}
```

# 矩阵快速幂

```
//这里是重载运算符的版本
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll mod=10;
struct Pot{
    int a[2][2]; //这里以2阶矩阵为例
    void init(){//单位阵
        for (int i=0;i<2;i++){//这个清空很重要 别忘了 不然会寄
            for (int j=0;j<2;j++){
                a[i][j]=0;
            }
        }
        for (int i=0;i<2;i++){
            a[i][i]=1;
        }
    }
    //因为可能遇到数据为 1e5 就是可能不是两个方阵相乘 所以我们要写一般形式
    vector<vector<int>> vec;
    void resiz(int n,int m){//这里也是相当于清空
        vec.resize(n,vector<int>(m)); //n*m -> vec[n][m]
    }
    void init(int n){
        for (int i=0;i<n;i++) a[i][i]=1; //化为单位阵
    }
};
//这里是重载运算符
Pot operator * (const Pot &A,const Pot &B){
    Pot C;
    for (int i=0;i<2;i++){//这里的清空也是 不然会寄
        for (int j=0;j<2;j++){
            C.a[i][j]=0;
        }
    }
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            for (int k=0;k<2;k++){
```

```
                C.a[i][j]+=A.a[i][k]*B.a[k][j]%mod; //如果要加%mod 就得加在这里 不然外面你还得在来一个%的重载运算符
            }
        }
    }
    return C; //这里一定要有返回值
}
Pot ksm(Pot x,int t){
    Pot Ans;
    Ans.init();
    while(t){
        if(t&1) Ans=Ans*x;
        x=x*x;
        t>>=1;
    }
    return Ans;
}
int main(){
    Pot D;
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            cin>>D.a[i][j];
        }
    }
    Pot E=ksm(D,34);
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            cout<<E.a[i][j]<<' ';
        }
        cout<<endl;
    }
    return 0;
}



//再来一个 不用重载运算符的版本(即用自定义函数)
#include <bits/stdc++.h>
using namespace std;
const int mod=10;
struct Pot{
    int a[2][2];
    void init(){
        for (int i=0;i<2;i++){
            for (int j=0;j<2;j++){
                if(i==j) a[i][j]=1;
                else a[i][j]=0;
            }
        }
    }
};
Pot jzc(Pot A,Pot B){//定义矩阵乘法
    Pot C;
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            C.a[i][j]=0;
        }
    }
```

```
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            for (int k=0;k<2;k++){
                C.a[i][j]+=A.a[i][k]*B.a[k][j]%mod;
            }
        }
    }
    return C;
}
Pot ksm(Pot x,int t){
    Pot Ans;
    Ans.init();
    while(t){
        if(t&1) Ans=jzc(Ans,x);
        x=jzc(x,x);
        t>>=1;
    }
    return Ans;
}
int main(){
    Pot D;
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            cin>>D.a[i][j];
        }
    }
    Pot E=ksm(D,34);
    for (int i=0;i<2;i++){
        for (int j=0;j<2;j++){
            cout<<E.a[i][j]<<' ';
        }
        cout<<endl;
    }
    return 0;
}
```

## *拓展欧几里得算法*

```
//辗转相除求最大公约数 (最简便的写法)
ll gcd(ll a, ll b){
    return b?gcd(b,a%b):a;
}
//补充 求最小公倍数
ll lcm(ll a, ll b){
    return a/gcd(a,b)*b;
}




typedef long long ll;
// 这个函数的返回值可以为int （即返回a b的最公约数）也可以为void
ll ext_gcd(ll a,ll b,ll& x,ll& y){//这里的x 和 y前面要加&是因为引用处理 即对他的值做实际的改变
```

```
    ll gcd=a;
    if(!b){
        x=1,y=0;
    }else {
        gcd=ext_gcd(b,(a%b),y,x);//建议从后面开始看起 因为 x=y1 y=x1-a/b*y1 (x1,y1为下一个状态的x y)
        y-=a/b*x; //这里用到 y=x1-a/b*y1
    }
    return gcd;
}
```

## 衍生 中国剩余定理(孙子定理)

```
//本质是对同余方程组使用 n-1次ext_gcd
ll Sunzi(ll *m,ll *a,int len){
    ll lcm = 1;
    ll ans = 0;
    for (int i=0;i<len;i++){
        ll k0,ki;
        ll d = ext_gcd(lcm,m[i],k0,ki);
        if ((a[i]-ans)%d!=0) return -1;
        else {
            ll t = m[i]/d;
            k0 = ( k0*(a[i]-ans)/d%t + t)%t;
            ans = k0*lcm + ans;
            lcm = lcm/d*m[i];
        }
    }
    return ans;
}
//应用通过同余方程组去求解x
```

## 欧拉筛

```
ll getPrime(ll n,bool vis[],ll prime[]){ //这里的prime数组是一个栈 用来存素数 vis用来记录它是不是素数
    ll tot=0;
    for(ll i=1;i<=n;i++)vis[i]=0;//0表示是素数 当然1是除外的 但是这里先算是
    for(ll i=2;i<=n;i++){
        if(!vis[i])prime[tot++]=i;
        for(ll j=0;j<tot;j++){
            if(prime[j]*i>n)break;
            vis[prime[j]*i]=1;//1表示不是素数
            if(i%prime[j]==0)break;//所根据的原来 就是一个数肯定是被它的最小因数筛掉了
        }
    }
    return tot;
}
```

## 欧拉函数

```
派(n) 表示 小于n的素数的个数
phi(n) 表示 小于等于n的与n互素的数的个数
phi(n=p1^a1+p2^a2+...+pt^at)=(1-1/p1)*...*(1-1/pt)*n (其中的n和前面一样)
typedef long long ll;
//欧拉函数和素数筛的应用 题目 在素数筛的同时 求出1~n的欧拉函数值
//phi[] 欧拉函数 prime[] 素数 vis[] 标记
int phi[1000],prime[1000],vis[1000];
int euler (int n){
    int cnt=0; //栈顶
    phi[1]=1;//特判 (规定)
    for (int i=2;i<=n;i++){
        if(!vis[i]){
            prime[cnt++]=i;
            phi[i]=i-1;//为什么呢？ 因为此时i为素数 那它的欧拉函数值肯定为n-1
        }
        for (int j=0;j<cnt&&prime[j]*i<=n;j++){
            vis[i*prime[j]]=1;//1表示不是素数 0是素数
            if(i%prime[j]==0){
                phi[i*prime[j]]=prime[j]*phi[i];//根据phi(p^a)=p*phi(p^(a-1))
                break;
            }else phi[i*prime[j]]=phi[i]*phi[prime[j]];//根据phi(n*m)=phi(n)*phi(m)
        }
    }
}
```

# 动态规划(dp)

## 最长上升公共子序列(LIS)

```
//第一种 n2的做法
定义状态 dp[i]:以第i个元素结尾的上升子序列的长度
    第一次for往后遍历 然后每遇到一个数就去判读
    状态转移方程 max(dp[k]) 1<=k<=i;
//第二种 nlogn的做法
定义状态 dp[i][j]:表示前i个元素里选长度为j的上升子序列中末尾元素的最小值（没有的先赋成无穷大）
    （然后可以优化掉一维） -> dp[j];
    状态转移方程 for往后遍历 然后每次遇到一个数 在dp里面找到比它小的所有值里面的最大值(二分去找 也可以说是去找到第一个比它小的值 把它换掉)
    二分找的的位置计为k
    如果找的到则dp[k+1]=i;
    如果找不到dp[k]=min(dp[k],a[i]); //这里的找不到 是指 遇到 无穷大这种 不含那种 小于起始点的情况
边界情况
    dp[0]=a[0];
```

### 模板

```
#include <bits/stdc++.h>
using namespace std;
const int N=1e5+10;
const int INF=0x3f3f3f3f;
```

```cpp
int n;
int a[N];
int dp[N];
int main(){
    cin>>n;
    for (int i=1;i<=n;i++) cin>>a[i];
    memset(dp,INF,sizeof(dp)); //记住memset是按字节分配的
    dp[1]=a[1];
    for (int i=1;i<=n;i++){
        int l=1,r=n,mid;
        while(l<=r){
            mid=l+r>>1;
            if(dp[mid]>=a[i]) r=mid-1;
            else l=mid+1;
        }
        dp[l]=a[i];
    }
    int res=1;
    for (int i=1;i<=n;i++){
        if(dp[i]==INF) break;
        res++;
    }
    cout<<res-1<<endl;
    return 0;
}
```

优化版

```cpp
//这样优化 相当于 我不用一开始就全部赋值 无穷大
//另外也不用再去 for 一遍 找那个最大的长度了
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e6+10;
int n;
int a[N];
int dp[N];
int main(){
    cin>>n;
    for (int i=1;i<=n;i++) cin>>a[i];
    dp[1]=a[1];
    int len=1;
    for (int i=1;i<=n;i++){
        if(dp[len]<a[i]) dp[++len]=a[i];
        else {
            int l=1,r=len,mid; //这里的二分终点变了
            while(l<=r) {
                mid= l + r >> 1;
                if(dp[mid]>=a[i]) r=mid-1;
                else l=mid+1;
            }
            dp[l]=a[i];
        }
    }
    cout<<len<<endl;
    return 0;
}
```

# 最长公共子序列(LCS)

状态定义 dp[i][j]表示a中前i个元素与b中前j个元素的最长公共子序列的长度
    状态转移分a[i]与b[j] 是否相等来写
    if(a[i]==b[j]) dp[i][j]=dp[i-1][j-1]+1; 这个是因为a[i]=b[j] 去掉哪一个都一样
    else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
然后写两个for去遍历
    边界情况
    dp[i][0]=dp[0][i]=0;  因为一个序列和一个空序列的配对数肯定为0

# 最大子段和

定义状态 dp[i]:以a[i]结尾的最大字段和
    状态转移方程 (分两种情况 1.包括a[i-1] 2.不包括a[i-1])
    dp[i]=max(dp[i-1]+a[i],a[i])
边界情况
    dp[0]=a[0];

# 最大子矩阵

思路:将这个矩阵按列分块（可用前缀和） 然后就变成上面那个问题了
    具体的做法就是 枚举行 然后加上 上面那个思路 (两个for)
    然后枚举的时候记录最大值

# 编辑距离

给定两个字符串a和b，现在可以对a在任意位置进行添加字母、删除字母或者把一个字母改成另外一个字母。问把a修改成b的最小操作次数是多少

状态定义 dp[i][j] 表示a中第i个与b中第j个满足题目要求需要操作的次数(a中的前i个改成b中的前j个需要的最小次数)
    状态转移 if(a[i]==b[i]) dp[i][j]=dp[i-1][j-1];
            else dp[i][j]=min(dp[i-1][j-1]+1,dp[i-1][j]+1,dp[i][j-1]+1); 包括 增删改
    //第二种写法
     讨论了一下i和j哪个大
     i大的话就不需要做添加操作
     如果i小的话就不需要做删除操作
     边界情况
     dp[i][0]=dp[0][i]=i; //还是比较好懂的

## 统计给定的子序列数量

给一个字符串s（长度2e5），问该字符串中，有多少个长度为5的子序列是happy（只要两个子序列，字符下标有一个不一样就算作不同）

```
状态定义
    dp[i][j]表示 a中前i个组成了b(happy)中前j个的方案数(key)
    状态转移
    dp[i][1]=dp[i-1][1]+dp[i-1][0]+1 a[i]=='h';
//因为是第一个 h 所以要加1
    dp[i][2]=dp[i-1][2]+dp[i-1][1] a[i]=='a';
    //为什么是dp[i-1][2]+dp[i-1][1]这种结构呢
    //因为遇到a的时候 状态可以从 原本有a和原本没有a转移过来
    //相当于 原本有a的本来就能构成 那方法数自然要转移过来 如果没a那刚好 我这个位置是a所以也要转移过来
    dp[i][3]=dp[i-1][3]+dp[i-1][2] a[i]=='p';
    dp[i][4]=dp[i-1][4]+dp[i-1][3]
    //为什么这里是两个方程呢 因为你遇到p的时候 可能是第3位也可能是第4位 所以两个都要更改
    dp[i][5]=dp[i-1][5]+dp[i-1][4] a[i]=='y'
    //那如果不是目标要求的字母呢(即不是happy中的字母)
    dp[i][1]=dp[i-1][1];
    dp[i][2]=dp[i-1][2];
    dp[i][3]=dp[i-1][3];
    dp[i][4]=dp[i-1][4];
    dp[i][5]=dp[i-1][5];
边界情况
    dp[i][0]=dp[0][1]=0;
```

# 背包dp

## 01背包

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=2e5+10;
//注意仔细看题 是 01dp 不是完全dp
//此题看似有点不好做 因为还有定义状态的时候 没有价值这一维 但是我们为什么不能自己创造呢
//题目问使空间最小 也就是求我们能够形成的最大空间
//那我们就让此题的每个物品的价值就等于它的体积
//定义状态 dp[i][j] 前i个物品 消耗了j个空间所能产生的最大价值(体积) 看起来很怪
int V,n;
int dp[N],w[N],v[N];
int main(){
    cin>>V>>n;
    for (int i=1;i<=n;i++){
        cin>>w[i];
        v[i]=w[i];
    }
    for (int i=1;i<=n;i++){
        for(int j=V;j>=w[i];j--){
            dp[j]=max(dp[j],dp[j-w[i]]+v[i]);
        }
    }
```

```
    cout<<V-dp[V]<<endl;
    return 0;
}
```

## 完全背包

对比01背包也就是一个物品可以选无数次

其定义状态是不变的 但是其状态转移方程是要改变的

dp [i] [j]=max(dp [i-1] [j],dp [i] [j-w[i]]+v[i])  //还是需要着重理解一下

但是这里的优化 同样是可以优化掉第一维 但是这里的第二个for就是正向的 而不是逆序 原因可以看上面

## 多重背包

```
const int N=1e5+10;
int c[N];//用来记每种物品的数量
int w[N];//用来记每种物品的重量
int dp[N];//用来存答案 也即上面定义的状态
void multi_knapsack1(int n,int w){//n表示有n种物品 w表示我的背包的最大容量
    int i,j,k;
    for ( i=1;i<=n;i++){
        for ( k=1;k<=c[i];k++){
            for ( j=w;j>=w[i];j--){
                dp[j]=max(dp[j-w[i]],dp[j]);
            }
        }
    }
}
//但是我们这种暴力 3个for 容易超时 是o(n3) 所以得优化
所以采用二进制拆分 //为什么可以用二进制拆分呢 因为 比如 1000 用二进制拆分 1 2 4 8 16 32 64 128 256 489 这样就可以表示完1000以内的所有数
//而且为什么需要去优化呢 就比如你要选两个物品 但是你暴力拆分 你就会 a[1] a[2]计算一次 a[2] a[3] 也会计算一次 所以时间复杂度就很高 所以就采用二进制优化
//多重背包
int w[N],c[N],v[N];
int dp[N];
void multi_beibao(){
    int tmpw[N];//每个物品实际的重量 与价值 因为后面我们要二进制展开 所以要先记录
    int tmpv[N];
    int k=0;//记录展开后的标号(行数) 因为你原本是10行 但是展开之后就不是 所以行数肯定又变化
    // 读入物品个数 背包容量 N  W
    for (int i=1;i<=N;i++){
        cin>>tmpw[i];
    }
```

```
    for (int i=1;i<=N;i++){
        cin>>tmpv[i];
    }
    //下面开始二进制展开
    for (int i=1;i<=N;i++){
        cin>>c[i];
        for (int j=1;j<=c[i];j<<=1){
            k++;
            w[i]=tmpw[i]*j;
            v[i]=tmpv[i]*j;
            c[i]-=j;
        }
        if(c[i]!=0){
            k++;
            w[i]=tmpw[i]*c[i];
            v[i]=tmpv[i]*c[i];
        }
    }
    //最后再调用正常的01背包
    beibao(k,W);
}
void beibao(int n,int w){
    for (int i=1;i<=n;i++){
        for (int j=w;j>=w[i];j--){
            dp[j]=max(dp[j-w[i]]+v[i],dp[j]);
        }
    }
}
```

## 混合背包

```
for (循环物品种类) {
  if (是 0 - 1 背包)
    套用 0 - 1 背包代码;
  else if (是完全背包)
    套用完全背包代码;
  else if (是多重背包)
    套用多重背包代码;
}
```

## 二维费用背包

比如说物品不但有体积限制，还有价格限制，这种有两维费用的背包就是二维费用背包，和一维背包没有本质上的差别

本质还是01背包就是在状态转移 以及遍历的时候 我们需要加上 价格这一个状态

```
for (int i=1;i<=n;i++){
        for (int j=m;j>=w[i];j--){
            for (int k=t;k>=p[i];k--){
                dp[j][k]=max(dp[j][k],dp[j-w[i]][k-p[i]]+v[i]);
            }
        }
    }
//也可以这样理解 (来着oi wiki)
for (int k = 1; k <= n; k++)
  for (int i = m; i >= mi; i--)    // 对经费进行一层枚举
    for (int j = t; j >= ti; j--)  // 对时间进行一层枚举
      dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
```

# 求方案数

题目为 我有 M块钱 然后我可以买好多东西 每个东西只有一份
求我把M块化光有几种方案

```
//所以这里的状态定义 dp[i] 表示化了i块钱所产生的点菜的方案数
//状态转移方程 dp[i]=dp[i]+dp[i-x]; x为当前菜品所需的钱 但是得倒序做 你比如你花10块你肯定得从前面转移过来 详细见代码
int n,m,x,dp[N];
int main(){
    cin>>n>>m;//n为菜品数 m为剩余的钱 也就是我的总钱数
    dp[0]=1; //我一块钱都不花 默认算一种方案 就是什么都不吃
    for (int i=1;i<=n;i++){
        cin>>x;//读入当前菜品的价格
        for (int j=m;j>=x;j--){
            dp[j]=dp[j]+dp[j-x];
        }
    }
    cout<<dp[m];
    return 0;
}
```

当然这里是每种菜品只有一种的做法 有点类似 01dp 但是当我的菜品数量无限时 也就是变成了类似完全dp了 此时也就是要正序做

```
//好现在题目变一下
```

有q个询问，每次要求删去第di个物品 求每次的方案数为多少？
也就是我ban掉其中一个
最简单 我把这个物品不算 我从头暴力来一遍 但是这样的时间复杂度是吃不消的 所以我得做优化
怎么优化 回退也即回溯
但是这个回退却是要正序做 原因 后面加的是前面的原先值 要先把前面还原 不然我每次相减不都是0了吗
具体操作看代码

```
int q=0;
    cin>>q;
    for (int i=1;i<=q;i++){
        cin>>x;
        for(int j=0;j<=m;j++){
            dp1[j]=dp[i];//为什么这里还要多开一遍数组呢 是因为我有多次询问
        }
        for (int j=x;j<=m;j++){
            dp1[j]=dp1[j]-dp1[j-c[x]]; //这里的c[x] 是存了x序号的物品的价格
```

```
        }
        cout<<dp1[m]<<endl;
    }
//但是 思考一下为什么可以减 是因为我这个菜品的顺序是无关的 你比如题目给的顺序 是 2 4 3 但是我自己认为是2 3 4有影响吗 （这里指ban掉4）
```

# 树形dp

树形DP，即在树上进行的 DP。由于树固有的递归性质，树形DP一般都是递归进行的

一般是从叶子节点开始，递归往上求解，即边界条件为叶节点，需要求的一般为根节点 //说是这么说 但是一般都是先dfs到底 然后再递归上来这个过程才是状态转移的时候

设的状态一般是以该节点为根子树的状态

关于树形dp的代码 我们以例题的形式讲解

```
eg1:
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e5+10;
//定义状态 dp[N][2]
//dp[x][0]表示不选x结点，以x为根的子树的最大快乐指数 (key)
//dp[x][1]表示选择x结点，以x为根的子树的最大快乐指数
//状态转移方程
//如果不选x，那么他的儿子（下属）可以选也可以不选。
//如果选了x，那么他的儿子（下属）就只能不选。
int dp[N][2];
int n,l,k;
int a[N];//记录每一个员工的快乐指数
int b[N];//来记录它是不是根节点
vector<int> tr[N];
void dfs(int f){
    dp[f][0]=0;
    dp[f][1]=a[f];
    for (int i=0;i<tr[f].size();i++){
        int to=tr[f][i];
        dfs(to);
        dp[f][0]+=max(dp[to][0],dp[to][1]);//这里对于了两种对于的状态转移方程
        dp[f][1]+=dp[to][0];//为什么这里是加等于呢 因为1个父节点有好多个儿子节点 那他们的快乐指数不是就是应该加起来吗
    }
}
int main(){
    cin>>n;
    for (int i=1;i<=n;i++){
        cin>>a[i];
    }
    for (int i=1;i<n;i++){
        cin>>l>>k;
        tr[k].push_back(l);
        b[l]=1;//代表它有父节点
    }
    int fa=0;
```

```
    for (int i=1;i<=n;i++){
        if(b[i]==0) fa=i;
    }
    dfs(fa);
    cout<<max(dp[fa][0],dp[fa][1])<<endl; //最后输出的时候别忘了比较一下 是选还是不选 那个大
    return 0;
}


eg2:
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e5+10;
int n,k,l,r;
vector<int> tr[N];
int dp[N][2]; //状态定义为 0表示当前这个节点选了 1则反正 然后dp都是用来记士兵数
void dfs(int x,int fa){//为什么要传一个fa进来 因为这是无根树 而且是无向边 所以要避免 走到儿子又走回父亲节点去
    dp[x][0]=1;
    dp[x][1]=0;
    for (int i=0;i<tr[x].size();i++){
        int to=tr[x][i];
        if(to==fa) continue; //如上面所述 需要特判的地方
        dfs(to,x);
        dp[x][0]+=min(dp[to][1],dp[to][0]);
        dp[x][1]+=dp[to][0];
    }
}
int main(){
    cin>>n;
    for (int i=1;i<=n;i++){
        cin>>l>>k;
        for (int i=1;i<=k;i++){
            cin>>r;
            tr[l].push_back(r);
            tr[r].push_back(l);
        }
    }
    dfs(0,-1);
    cout<<min(dp[0][0],dp[0][1])<<endl;
    return 0;
}


eg3:
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll N=1e5+10;
const ll mod=1e9+7;
//这题的状态定义以及状态转移方程是关键
ll dp[N][4];// 表示当前节点涂某种颜色的方案数
ll vis[N];//表示是否已经涂过色了
ll n,k,l,r;
vector<ll> tr[N];
void dfs(ll x,ll fa){
    if(vis[x]==0){
```

```
        dp[x][1]=1;
        dp[x][2]=1;
        dp[x][3]=1;//这里赋值成1 也可以理解为是为了下面乘法 你总不能是0把 0*任何数都为0 那还有什么好做的
    }
    for (ll i=0;i<tr[x].size();i++){
        ll to=tr[x][i];
        if(to==fa) continue;
        dfs(to,x);
        //重中之重 如何转移
        dp[x][1]=dp[x][1]*(dp[to][2]+dp[to][3])%mod;//为什么是乘法 是后面两个相加这个很好理解
        dp[x][2]=dp[x][2]*(dp[to][3]+dp[to][1])%mod;//因为我的左儿子有5种方案 而我的右儿子有 3种 那我这个父节点有几种
        dp[x][3]=dp[x][3]*(dp[to][2]+dp[to][1])%mod;//3*5
    }
}
int main(){
    cin>>n>>k;
    for (int i=1;i<n;i++){
        cin>>l>>r;
        tr[l].push_back(r);
        tr[r].push_back(l);
    }
    for (int i=1;i<=k;i++){
        int a=0,b=0;
        cin>>a>>b;
        dp[a][b]=1;
        vis[a]=1;
    }
    dfs(1,0);
    cout<<(dp[1][1]+dp[1][2]+dp[1][3])%mod<<endl;
    return 0;
}
```

# STL

## *STL容器*

```
vector(不定长数组)
    o(1)
    定义 vector<type> v[];//开二维加后面那个[]
    插入（从最后插） v.push_back(x);
    删除（最后） v.pop_back(x);
    o(n)
    获取长度 v.szie();
    清空 v.clear();
    头元素指针 v.begin(); 尾指针 v.end()-1;
set(集合)
    自己从小到大排好序//同理也是可以开二维
    set(去重) multiset(不去重)
    定义 (multi) set<type> s;
    o(logn)
    插入 s.insert(x);
    删除 s.erase(x);
```

```
    查询 s.find(x) 如果不存在，返回 s.end()
    o(n)
    清空 s.clear();
    set的遍历用auto i:s;
map(映射)
    定义 (unordered_)map<type1,type2> mp; o(log) //不排序映射 unordered_map o(1)
    使用和数组差不多
    清空 mp.clear(); o(n)
stack(栈)//可用数组模拟回顾py
    定义 stack<type> st;
    压入新元素 st.push(x);
    出栈 st.pop();
    栈顶 st.top();
    是否栈空 st.empty();
queue(队列)//可用数组模拟回顾py 数组模拟我就可以双端操作了（头尾）拓展 循环队列（后期补）
    定义 queue<type> q;
    查询头元素 q.front();
    查询尾元素 q.back();
    入队 q.push(x);
    出队 q.pop();
    队是否空 q.empty();
    一些函数：

priority_queue(优先队列)
    定义 priority_queue<type> q;
    o(log)
    查询队头(优先级最高) q.top();
    入队 q.push(); //会自己排好序 （默认降序）
    出队 q.pop();
    查询是否为空 q.empty();
    //如何升序 priority_queue<int,vector<int>,greater<int>>q;
```

# *STL函数*

sort
next_permutation
lower_bound() upper_bound() //重点 和vector集合怎么写
lower_bound( )和upper_bound( )都是利用二分查找的方法在一个排好序的数组中进行查找的。

在从小到大的排序数组中，(不递减的数组里)
lower_bound( begin,end,num)：从数组的begin位置到end-1位置二分查找第一个大于或等于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。
upper_bound( begin,end,num)：从数组的begin位置到end-1位置二分查找第一个大于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。

在从大到小的排序数组中，重载lower_bound()和upper_bound()
lower_bound( begin,end,num,greater<type>() ):从数组的begin位置到end-1位置二分查找第一个小于或等于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。
upper_bound( begin,end,num,greater<type>() ):从数组的begin位置到end-1位置二分查找第一个小于num的数字，找到返回该数字的地址，不存在则返回end。通过返回的地址减去起始地址begin,得到找到数字在数组中的下标。
全排列(按字典序) next_permutation(a,a+n) //a为数组的首地址 比当前数组的顺序还大的全排列全输出 vector则用 v.begin() v.end()

# 搜索

## *二分查找*

关于二分的版本有很多的 最基础的版本就是 l+r>>1

然后3个if 满足 则返回 不满足 要么 l=mid+1 r=mid-1

下面介绍这种的推广应用

例题1

这种写法的一个好处就是 我里面不会出现像下面第二种 对应 不同的要 偏左还是偏右 我全部都是 r=mid-1 l=mid+1 至于if语句里等号 放哪 你仔细想一下 就行 同时答案也是一起思考的 是到底返回l还是r

eg: 找到x第一次出现的位置 (元素按升序)

```
int l=begin,r=end;
while(l<=r){
    int mid=l+r>>1;
    if(a[mid]>=x) r=mid-1;
    else l=mid+1;
}
return l
```

这一版的二分查找与二分答案一致

# *BFS(广搜)*

bfs是和队列结合的 然后记住一个 广搜 搜到即最短

基本上 是 和 迷宫 图 数轴 (可转化为图 )等 结合

这题 就是数组 转 图的方式

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=2e5+10;
ll n;
ll a[N];// a[i] 用来记能量波动
ll dis[N];
ll vis[N];
vector <int> G[N];
queue <int> q;
map <ll,ll> mp;
int bfs(){
    while(q.size()){
        ll now=q.front();
        q.pop();
        for (auto v:G[now]){
            if (v==n){
                return dis[now]+1;
            }
            if(vis[v]){
                continue;
            }
            vis[v]=1;
            dis[v]=dis[now]+1;
            q.push(v);
        }
    }
    return 0;
}
int main(){
    cin>>n;
    for (int i=1;i<=n;i++){
        cin>>a[i];
        if(i!=n){
            G[i].push_back(i+1);
            G[i+1].push_back(i);//邻链表 存图
        }
        if (mp.count(a[i])){
            G[mp[a[i]]].push_back(i);
        }
        mp[a[i]]=i;
    }
    q.push(1),vis[1]=1;dis[1]=0;
    cout<<bfs()<<endl;
    return 0;
}
```

# *DFS(深搜)*

一般是和栈结合 但是我们一般是写成递归

这里给一个迷宫类型的深搜 同时 关于深搜的一个应用是 全排列 然后有一个对应的stl函数 在下面列出

```
#include <bits/stdc++.h>
using namespace std;
int n,m,t;
int sx,sy,fx,fy;
int res;
int ans[10][10];
int vis[10][10];
int dx[4]={1,-1,0,0};
int dy[4]={0,0,1,-1};
void dfs(int x,int y){
    if(x==fx&&y==fy){
        res++;
        return ;
    }
    for (int i=0;i<4;i++){
        int X=x+dx[i];
        int Y=y+dy[i];
        if(X>=1&&X<=n&&Y>=1&&Y<=m&&vis[X][Y]==0&&ans[X][Y]==0){
            vis[X][Y]=1;
            dfs(X,Y);
            vis[X][Y]=0;
        }
    }
}
int main(){
    cin>>n>>m>>t;
    cin>>sx>>sy>>fx>>fy;
    for (int i=1;i<=t;i++){
        int x,y;
        cin>>x>>y;
        ans[x][y]=1;//代表是障碍物
    }
    vis[sx][sy]=1;//起点也是要标记的 不然往前走 又走回来了
    dfs(sx,sy);
    cout<<res<<endl;
    return 0;
}
```

# 字符串

## *字符串哈希*

首先先来解释一下什么是哈希

哈希是一种映射操作 相当于你可以把一个很大很大的数映射成一个很小很小的数 例如你可以把10进制的大数 映射成 131进制的数

当然这种映射是自己想的

字符串哈希就是运用这个思想 我两个字符串去比较大小 要一个一个比较 效率太慢了 所以可以采用字符串哈希 然后映射之后的值 一样 那就说明这两个字符串一样

(注意哈希值相同的时候 可能字符串还是不同 为什么 因为我们映射之后 可能这个进制的 幂次会很大 比如你 131的5次幂 10次幂 可能会爆long long 所以我们会进行取模操作 所以这样就会导致可能存在两个字符串不同 但是哈希值相同 这我们叫做**哈希冲突**，所以有时我们为了避免就要去较好的进制数base 和取模数mod 以为必要时实现双哈希 )

具体可见oi-wiki

这里我们通过一道板子题来讲解基础的字符串哈希

本题的思想是 用哈希值来代表字符串 加速判断

这就有一个疑问了 我是多次询问 区间 那我总不能每次都从头来吧 那不是和暴力没什么差别了？ 这时就想到了 前缀和 那我能不能也搞一下哈希前缀和

hash[L] 记录从第一个位置 到第L个位置的哈希值 hash[R]同理

那怎么计算 hash[L,R]呢

（贴张公式上去）

代码翻译如下

(hash1[r]-hash1[l-1]*p[r-l+1]%mod+mod)%mod;

但是字符串哈希只能判断两个字符串是否一样

不一样 却不能输出谁大谁小

所以还需要二分 (为什么可以二分呢 而且我们这里不是严格的二分查找 而是二分答案 去二分长度 看我从)

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=2e5+10;
//较好的base 131 233
ll base=131;
//较好的mod
ll mod=998244353;
ll p[N];//用来存p进制的几次幂 P[0]=1 p[1]=base p[2]base*base
ll hash1[N];//用来存前缀哈希值
ll hash2[N];
ll n,m,q;
ll l1,l2,r1,r2;
string s1,s2;
int query1(ll l,ll r){
```

```
    return (hash1[r]-hash1[l-1]*p[r-l+1]%mod+mod)%mod;//根据前面的思想写出来的代码式子
}
int query2(ll l,ll r){
    return (hash2[r]-hash2[l-1]*p[r-l+1]%mod+mod)%mod;
}
int main(){
    cin>>n>>m>>q;
    cin>>s1>>s2;
    p[0]=1;
    for (int i=1;i<=N;i++){
        p[i]=p[i-1]*base%mod;
    }
    for (int i=1;i<=n;i++){
        hash1[i]=(hash1[i-1]*base%mod+s1[i-1])%mod;
    }
    for (int i=1;i<=m;i++){
        hash2[i]=(hash2[i-1]*base%mod+s2[i-1])%mod;
    }
    while(q--){
        cin>>l1>>r1>>l2>>r2;
        //开始二分长度(答案)
        ll l=0,r=r1-l1; //其实式长度的增量 因为你第一个位置也是要判断的
        while(l<=r){
            ll mid=l+r>>1;
            if(query1(l1,l1+mid)==query2(l2,l2+mid)){
                l=mid+1;
            }else r=mid-1;
        }
        if(l==r1-l1+1) cout<<"=\n";//如果比我长度还大 那不就说明全都相等吗
        else if (s1[l1+l-1]<s2[l2+l-1]) cout<<"<\n";
        else cout<<">\n";
    }
    return 0;
}
```

## *KMP*

# 杂项

## *组合数学*

### 盒与小球

将球设为nn个，盒子设为mm个

**1.球相同，盒子不同，无空盒**

隔板法

$$\binom{n-1}{m-1}$$

**2.球相同，盒子不同，有空盒**

$$\binom{n+m+1}{m-1}$$

**3.球相同，盒子相同，有空盒**